Date: 04/02/2025

1. Agile

Agile is a **flexible and fast way of working** in software development and project management. Instead of planning everything in advance, teams work in **small steps**, get **continuous feedback**, and make **improvements along the way**.

Key Features of Agile:

✅ **Work in Small Steps** – Projects are divided into smaller tasks (iterations or sprints).
✅ **Customer Involvement** – Regular feedback is taken from customers.
✅ **Quick Changes** – Teams can adjust plans as needed.
✅ **Team Collaboration** – Developers, testers, and business teams work closely together.
✅ **Faster Delivery** – Software is delivered in small parts instead of waiting for the full product.

Example - (Online Shopping Website)

□ **Initial Release** – Develop basic features (e.g., product browsing, checkout) in 2-4 weeks.

□ **Customer Feedback** – Get real user feedback.

□ **Improve** – Add features like payment, reviews, and recommendations.

□ **Iterate** – Continue improving based on ongoing feedback.

□ **Quick Updates** – Regular updates make the site better over time.

Advantages

Saves Time

Reduces Risk

2. Scrum

Scrum is a specific Agile framework used to manage and complete projects in small, manageable steps. It is widely used in software development but can be applied to other fields as well. Scrum focuses on teamwork, accountability, and delivering high-value results quickly.

Key Elements of Scrum:

1. **Sprint**: A time-boxed period (usually 2-4 weeks) during which a set of tasks is completed.
2. **Scrum Team**: Includes roles like **Product Owner**, **Scrum Master**, and **Development Team**.

3. **Backlog**: A prioritized list of tasks that need to be completed.
4. **Daily Stand-up**: A quick daily meeting where the team discusses progress, challenges, and plans for the day.
5. **Sprint Review**: A meeting at the end of each sprint to demonstrate the completed work.
6. **Sprint Retrospective**: A meeting where the team reflects on the sprint and discusses improvements for the next one.

Scrum Roles

1. **Product Owner**: Manages the backlog and ensures the team is working on the most valuable tasks.
2. **Scrum Master**: Facilitates the process, removes obstacles, and ensures the team follows Scrum practices.
3. **Development Team**: A group of professionals who work together to deliver the project increment.

Real-Life Scrum Example (Fitness App):

1. **Sprint Planning**: Build basic features like **user registration** and **step counter** in 2-4 weeks.
2. **Work During Sprint**: The team works on the features.
3. **Daily Stand-up**: Team members give brief updates on progress.
4. **Sprint Review**: Show the app's progress (registration, step counter) to stakeholders.
5. **Sprint Retrospective**: Reflect on what worked well and what to improve.

Next Sprint: Add exercise tracking and nutrition logs.

Why Scrum Works:

- **Quick Feedback** from users.
- **Continuous Improvement** with each sprint.

Sprint in Scrum

A **sprint** is a **short, fixed period of time** (usually 2-4 weeks) during which a team works to complete specific tasks or features. At the end of the sprint, the team delivers something **working** (a feature or product) to show progress.

Sprint Example:

Developing a Blog Website:

1. **Sprint Planning**: The team plans to develop the **homepage** and **blog post creation page** in a 2-week sprint.

2. **Work During Sprint**: The team works on the design, coding, and testing of these pages.
3. **Sprint Review**: At the end of the sprint, the team shows the completed **homepage** and **blog post creation page** to the client for feedback.

After the sprint, they plan the next sprint to work on features like **comment section** and **user login**.

Project Management

Project management is the process of planning, organizing, and overseeing tasks and resources to achieve a specific goal or complete a project. It involves managing people, time, and money to make sure everything is done on time, within budget, and to the desired quality.

Key Aspects of Project Management:

1. **Planning**: Defining the project's goals, tasks, timelines, and resources needed.
2. **Organizing**: Assigning tasks, roles, and responsibilities to team members.
3. **Execution**: Carrying out the project tasks and monitoring progress.
4. **Monitoring & Controlling**: Tracking the project's progress, handling issues, and making adjustments if needed.
5. **Closing**: Finalizing and delivering the project, ensuring all goals have been met.

Example:

For a **website redesign project**:

1. **Planning**: Decide on the scope of the website redesign (e.g., new layout, improved navigation, mobile-friendly design).
2. **Organizing**: Assign team members for design, development, and testing.
3. **Execution**: The team works on designing the site and coding the pages.
4. **Monitoring**: Track progress, make sure tasks are completed on time, and address any delays.
5. **Closing**: Launch the new website and make sure everything is working as planned.

UML (Unified Modeling Language)

UML is a **standardized visual language** used to design and model software systems. It helps developers, designers, and stakeholders understand the structure, behavior, and interactions of a system before it's built.

Key Points of UML:

1. **Visual Representation**: UML uses diagrams to represent different aspects of a system (e.g., class structure, interactions, behavior).
2. **Standardized**: It's a universally accepted notation, making it easier for teams to communicate ideas clearly.
3. **Flexible**: UML can be used for various software development methodologies (e.g., Agile, Waterfall).
4. **Multiple Diagrams**: UML consists of **different types of diagrams** to represent various aspects of a system (e.g., static structure, dynamic behavior, interactions).
5. **Modeling**: It helps visualize how the system will work, which is essential for designing complex systems.

Types of UML Diagrams:

1. **Class Diagram**: Shows the structure of the system, including **classes**, **attributes**, **methods**, and **relationships**.
2. **Use Case Diagram**: Describes the system's **functionalities** from the user's perspective.
3. **Sequence Diagram**: Shows the **sequence of interactions** between objects or components.
4. **Activity Diagram**: Represents the flow of **activities** and processes in the system.
5. **State Diagram**: Represents the **states** of an object and transitions between those states.

Example:

Class Diagram for a Library System:

1. **Class: Book**
   - **Attributes**: title, author, ISBN
   - **Methods**: checkOut(), returnBook()
2. **Class: Member**
   - **Attributes**: name, membershipID
   - **Methods**: borrowBook(), returnBook()
3. **Relationship**: A **Member** can **borrow** multiple **Books**, but a **Book** can only be borrowed by one **Member** at a time.

Coding Standards

Coding standards are a set of guidelines and rules that developers follow while writing code. These standards help improve **code readability**, **maintainability**, and **collaboration** among team members.

Key Points of Coding Standards:

1. **Consistency**: Ensures that code looks and behaves in a similar way across the project, making it easier for developers to understand each other's work.
2. **Readability**: Focuses on writing clean and easy-to-understand code so that others can easily read, modify, and maintain it.
3. **Best Practices**: Encourages the use of tried-and-tested methods for writing efficient, secure, and error-free code.
4. **Documentation**: Promotes the use of comments and docstrings to explain the code, especially complex logic or non-obvious behavior.
5. **Naming Conventions**: Defines rules for naming variables, functions, classes, and other elements to make the code meaningful and easy to understand.
6. **Code Formatting**: Specifies guidelines for indentation, spacing, and line length to ensure consistent formatting.

Common Coding Standard Guidelines:

1. **Indentation**: Use 4 spaces (or tabs) to indent code for better readability.
2. **Variable Names**: Use meaningful and descriptive names (e.g., userAge instead of x).
3. **Functions/Methods**: Name functions to clearly describe their purpose (e.g., calculateTotalAmount()).
4. **Comments**: Write clear and concise comments to explain what your code is doing, especially for complex logic.
5. **Avoid Hardcoding**: Don’t hardcode values directly into the code; use variables or constants.

Example (Python Code):

python

CopyEdit

```python
# Good coding standard example


def calculate_area(radius):
    """Function to calculate area of a circle"""
    if radius < 0:
        return "Invalid radius"
    return 3.14 * radius * radius
```

```
# Proper naming, indentation, and documentation
area = calculate_area(5)
print(area)
```

In this example:

- The function name calculate_area is descriptive.
- The indentation is consistent (4 spaces).
- There's a comment explaining the purpose of the function.

Coding Standards: PEP 8 and MOSCOW Principle

---

1. PEP 8 (Python Enhancement Proposal 8):

PEP 8 is the **official style guide** for Python code. It provides guidelines on writing clean, readable, and consistent code.

Key Points:

- **Indentation**: Use 4 spaces per indentation level.
- **Line Length**: Limit all lines to **79 characters**.
- **Naming Conventions**:
  - **Variables and functions**: use snake_case (e.g., my_function).
  - **Classes**: use CamelCase (e.g., MyClass).
  - **Constants**: use UPPERCASE_WITH_UNDERSCORES (e.g., MAX_LENGTH).
- **Whitespace**: Avoid extra spaces in expressions (e.g., x = 10 instead of x = 10).
- **Docstrings**: Use triple quotes for docstrings to describe functions and classes.

Example of PEP 8 Code:

python

CopyEdit

```
def calculate_area(radius):
    """Calculate the area of a circle."""
    pi = 3.14159
    return pi * radius ** 2
```

---

2. MOSCOW Principle (MoSCoW Method):

The MOSCOW Principle is a **prioritization technique** used in project management (especially in Agile). It helps to define the importance of various tasks, features, or requirements.

Key Points:

- **M**ust Have: Non-negotiable requirements. The project cannot proceed without them.
- **S**hould Have: Important but not essential requirements. They can be delayed but should be included if possible.
- **C**ould Have: Nice-to-have features that can be included if time and resources permit.
- **W**on't Have: Features that are agreed to be left out or deferred for the current project phase.

Example of MOSCOW in Software Development:

- **Must Have**: User login and security.
- **Should Have**: Product search functionality.
- **Could Have**: Dark mode for the user interface.
- **Won't Have**: Integration with social media (for this phase).

Summary:

- **PEP 8**: A set of coding standards for Python to ensure **clean, readable, and consistent code**.
- **MOSCOW**: A **prioritization method** to decide which project features are essential and which can be deferred.

SDLC (Software Development Life Cycle)

SDLC is a process used by software developers to design, develop, test, and maintain software applications. It breaks down the software development process into clear stages, ensuring the project is completed in a structured and efficient manner.

Key Stages of SDLC:

1. **Requirement Gathering**:
   Understand the client's needs and document the system requirements.
2. **System Design**:
   Plan the system’s architecture, user interface, and overall structure based on the requirements.
3. **Development**:
   Write the actual code and implement the design.

4. **Testing**:
   Ensure the software works correctly by finding and fixing bugs or errors.
5. **Deployment**:
   Release the software to the users or customers.
6. **Maintenance**:
   Provide ongoing support to fix bugs, make improvements, and update the system as needed.

Types of SDLC Models:

1. **Waterfall Model**: A **linear** and **sequential** approach where each stage must be completed before moving to the next.
2. **Agile Model**: A more **flexible** and **iterative** approach where software is developed in small chunks and improved based on feedback.
3. **V-Model**: Similar to Waterfall, but with a focus on testing each stage before moving to the next.
4. **Spiral Model**: Combines iterative development with regular risk assessment.

Example (Waterfall SDLC for an Online Store Website):

1. **Requirement Gathering**: The client wants a website with product listings, a shopping cart, and payment integration.
2. **Design**: The team designs the website layout, database structure, and payment flow.
3. **Development**: The team codes the website using HTML, CSS, JavaScript, and integrates the payment system.
4. **Testing**: The website is tested for bugs, compatibility, and security vulnerabilities.
5. **Deployment**: The website is launched for customers to use.
6. **Maintenance**: The team fixes any issues that arise and adds new features based on customer feedback.

Software Testing

Software testing is the process of checking if a software works as intended, by identifying bugs and ensuring quality.

Key Types:

1. **Manual Testing**: Tester checks the software manually.
2. **Automated Testing**: Uses tools to automatically run tests.
3. **Functional Testing**: Checks if the software's features work as expected.

4. **Non-Functional Testing**: Checks performance, security, and usability.
5. **Regression Testing**: Ensures new changes don't break existing functionality.
6. **Acceptance Testing**: Verifies if the software meets user needs.

Testing Phases:

1. **Unit Testing**: Tests individual components (e.g., a function).
2. **Integration Testing**: Tests how different parts of the system work together.
3. **System Testing**: Tests the entire system as a whole.
4. **User Acceptance Testing (UAT)**: Tests if the software meets user expectations.

Unit Testing

Unit testing is the process of testing individual components (or units) of a software application to ensure they work as expected.

Key Points:

1. **Isolate Functionality**: Test small, isolated parts of the system (e.g., adding/removing items in a cart).
2. **Test Cases**: Create specific test cases for different inputs and expected outcomes (e.g., adding an item, removing an item).
3. **Automation**: Use tools (e.g., JUnit, PyTest) to run tests automatically for faster feedback.
4. **Verify Expected Results**: Check if the actual behavior matches the expected behavior (e.g., cart shows correct items and quantities).

Example:

- **Add item**: Add an item to the cart, check if the item appears with correct quantity.
- **Remove item**: Remove an item, verify it’s no longer in the cart.

Purpose:

- Ensures the functionality works correctly.
- Catches bugs early in development.
- Makes code easier to maintain.

Error Handling:

Error handling involves managing issues that arise during program execution to prevent crashes and ensure smooth operation.

Key Points:

- **Try-Except**: Catch errors and handle them appropriately.
- **Graceful Degradation**: Allow the program to continue running even after errors.
- **Logging**: Keep track of errors for later review.

Example:

python

CopyEdit

```
try:
    result = 10 / 0  # Error
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Risk Management:

Risk management involves identifying, assessing, and dealing with potential issues that might affect the project’s success.

Key Points:

- **Identify Risks**: Recognize possible issues.
- **Assess Impact**: Determine their likelihood and consequences.
- **Mitigate**: Create plans to reduce or avoid risks.

Example:

- **Risk**: Vendor delays.
- **Mitigation**: Use a backup vendor or adjust timelines.

Summary:

- **Error Handling**: Prevents crashes by managing unexpected issues in the code.
- **Risk Management**: Identifies and reduces potential risks that could affect project outcomes.

What is a Docstring in Python?

A **docstring** (short for **documentation string**) is a special kind of string used to document a module, function, class, or method in Python. It helps developers understand the purpose and usage of the code.

Key Features of Docstrings:

1. **Written as a string inside triple quotes (""" """ or ''' ''').**
2. **Placed at the beginning of a function, class, or module.**
3. **Can be accessed using help(object).**
4. **Follows PEP 257 (Python Docstring Convention).**

Types of Docstrings & Examples

1. Function Docstring

Explains what the function does, its parameters, and return values.

python

CopyEdit

```
def add_numbers(a, b):
    """Adds two numbers and returns the result.

    Args:
        a (int): First number.
        b (int): Second number.

    Returns:
        int: Sum of the two numbers.
    """
    return a + b

print(add_numbers(3, 5))  # Output: 8
help(add_numbers)  # Displays the docstring
```

---

2. Class Docstring

Describes the purpose of a class and its attributes.

python

CopyEdit

```
class Car:
    """A simple Car class.
    Attributes:
        brand (str): The brand of the car.
        year (int): The manufacturing year of the car.
    """

    def __init__(self, brand, year):
        """Initializes the Car with a brand and year."""
        self.brand = brand
        self.year = year

my_car = Car("Toyota", 2022)
help(Car)  # Displays the class docstring
```

---

3. Module Docstring

Used at the top of a Python file to describe its purpose.

python

CopyEdit

```
"""This module provides utility functions for mathematical operations."""

def multiply(a, b):
    """Multiplies two numbers."""
    return a * b
```

Accessing Docstrings

You can view a function or class docstring using:

python

CopyEdit

```
print(add_numbers.__doc__)
help(add_numbers)
```

Best Practices (PEP 257)

✅ Keep docstrings concise but informative.
✅ Use **triple double quotes (""")**, even for one-line docstrings.
✅ Start with a short summary, followed by details if needed.
✅ Use indentation properly for multi-line docstrings.

Virtual Environment:

A **virtual environment** is an isolated workspace used to manage dependencies and packages for a specific project, without affecting the system-wide Python installation.

Key Points:

1. **Isolation**: Each virtual environment has its own set of libraries and dependencies, separate from the global Python environment.
2. **Dependency Management**: Helps avoid conflicts between different versions of packages used in different projects.
3. **Project-Specific**: You can create a virtual environment for each project, ensuring that the correct versions of dependencies are used.
4. **Easily Reproducible**: Others can set up the same environment using a requirements file (e.g., requirements.txt).

Benefits:

- **No Conflicts**: Keeps dependencies isolated, preventing version conflicts.
- **Easier Collaboration**: Team members can share the same environment setup.
- **Cleaner Setup**: No need to install packages globally.

Example (Using venv in Python):

1. **Create a virtual environment**:

bash

CopyEdit

```
python -m venv myenv
```

2. **Activate the virtual environment**:
   - On Windows:

bash

CopyEdit

```
myenv\Scripts\activate
```

   - On macOS/Linux:

bash

CopyEdit

source myenv/bin/activate

3. **Install packages**:

bash

CopyEdit

pip install requests

4. **Deactivate the environment**:

bash

CopyEdit

deactivate

Basics of Python

Python is a high-level, interpreted programming language known for its simplicity and readability. It's widely used for web development, data science, automation, and more.

Here's a quick overview of the basics:

---

1. Variables and Data Types

- **Variables** are used to store values.
- **Data Types** in Python include:
    - **Integers**: Whole numbers (5, -10)
    - **Floats**: Decimal numbers (3.14, 0.5)
    - **Strings**: Text enclosed in quotes ("Hello", 'Python')
    - **Booleans**: True or False values (True, False)
    - **Lists**: Ordered collection of items ([1, 2, 3])
    - **Tuples**: Immutable ordered collection ((1, 2, 3))
    - **Dictionaries**: Key-value pairs ({"name": "Alice", "age": 25})

2. Operators

- **Arithmetic Operators**: +, -, *, /, // (floor division), % (modulus), ** (exponentiation)
- **Comparison Operators**: ==, !=, >, <, >=, <=
- **Logical Operators**: and, or, not

3. Control Flow

- **If-Else Statements**: Conditional execution.

python

CopyEdit

```
if x > 5:
   print("Greater")
else:
   print("Less or equal")
```

- **Loops**:
  - **For loop** (for iterating over a sequence):

python

CopyEdit

```
for i in range(5):  # prints numbers 0 to 4
   print(i)
```

  - **While loop** (repeats while condition is true):

python

CopyEdit

```
while x < 5:
   x += 1
   print(x)
```

4. Functions

Functions are reusable blocks of code that perform specific tasks.

python

CopyEdit

```
def greet(name):
   return f"Hello, {name}"


result = greet("Alice")
print(result)
```

5. Lists and Loops

- **List**: A collection that can store multiple items.

python

CopyEdit

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

6. Classes and Objects

Python is **object-oriented**, meaning you can define **classes** and create **objects** from them.

python

CopyEdit

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def speak(self):
        return f"{self.name} says woof!"

my_dog = Dog("Buddy", "Golden Retriever")
print(my_dog.speak())
```

7. Importing Modules

Python has many built-in modules for tasks like math operations, file handling, etc.

python

CopyEdit

```
import math
result = math.sqrt(16)
print(result)
```

8. Error Handling

Handling errors using try and except to prevent program crashes.

python

CopyEdit

```
try:
    x = 10 / 0
```

```
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

9. Comments

Comments are used to explain code. Python ignores them during execution.

python

CopyEdit

```
# This is a single-line comment
```

For multi-line comments, use triple quotes:

python

CopyEdit

```
"""
This is a multi-line comment
spanning multiple lines
"""
```

DataFrame Operations in Python

A **DataFrame** is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It's a central data structure in **Pandas**, a popular data analysis library in Python.

Here are some common DataFrame operations:

---

1. Creating a DataFrame

You can create a DataFrame from various data structures like lists, dictionaries, or external data sources (CSV, Excel, etc.).

Example:

python

CopyEdit

```
import pandas as pd


# Creating DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'Los Angeles', 'Chicago']}
```

df = pd.DataFrame(data)

print(df)

---

2. Viewing Data

- **Head()**: Displays the first 5 rows of the DataFrame.

python

CopyEdit

print(df.head())

- **Tail()**: Displays the last 5 rows.

python

CopyEdit

print(df.tail())

- **Info()**: Displays information about the DataFrame (column types, non-null counts).

python

CopyEdit

print(df.info())

---

3. Accessing Data

- **By Column**: Use column names to access specific columns.

python

CopyEdit

print(df['Name']) # Access the 'Name' column

- **By Row**: Use .iloc[] (indexing) or .loc[] (label-based) for accessing rows.

python

CopyEdit

print(df.iloc[1]) # Access the second row

print(df.loc[1])   # Access row with index 1

---

4. Filtering Data

You can filter data based on conditions.

python

CopyEdit

```python
# Filter rows where Age is greater than 30
filtered_df = df[df['Age'] > 30]
print(filtered_df)
```

---

5. Adding and Removing Columns

- **Add a Column**:

python

CopyEdit

```python
df['Country'] = ['USA', 'USA', 'USA']
print(df)
```

- **Remove a Column**:

python

CopyEdit

```python
df.drop('Country', axis=1, inplace=True)  # axis=1 refers to columns
print(df)
```

---

6. Modifying Data

- **Modify Column Values**:

python

CopyEdit

```python
df['Age'] = df['Age'] + 1  # Increment age by 1
print(df)
```

---

7. Sorting Data

Sort data based on one or more columns.

python

CopyEdit

```python
# Sort by Age
df_sorted = df.sort_values(by='Age', ascending=False)
```

```
print(df_sorted)
```

---

8. Grouping Data

Group data based on certain columns and apply functions (e.g., sum, mean).

python

CopyEdit

```
# Group by 'City' and calculate the mean age
grouped_df = df.groupby('City')['Age'].mean()
print(grouped_df)
```

---

9. Merging DataFrames

You can merge two DataFrames on a common column.

python

CopyEdit

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [1, 2, 4], 'City': ['NY', 'LA', 'SF']})

merged_df = pd.merge(df1, df2, on='ID', how='inner')  # Inner join
print(merged_df)
```

---

10. Handling Missing Data

- **Detect missing values**:

python

CopyEdit

```
print(df.isnull())
```

- **Fill missing values**:

python

CopyEdit

```
df['Age'].fillna(df['Age'].mean(), inplace=True)  # Fill NaN values with the mean
```

- **Drop rows with missing values**:

python

CopyEdit

df.dropna(inplace=True)

---

11. Aggregating Data

Use **agg()** to apply multiple aggregation functions at once.

python

CopyEdit

agg_df = df.agg({'Age': ['mean', 'sum'], 'City': 'count'})

print(agg_df)

---

12. Saving DataFrames

You can save DataFrames to various file formats like CSV, Excel, etc.

python

CopyEdit

# Save DataFrame to CSV

df.to_csv('data.csv', index=False)

# Save DataFrame to Excel

df.to_excel('data.xlsx', index=False)

OOPs (Object-Oriented Programming) Concepts

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data in the form of fields (attributes) and code in the form of methods (functions). Here are the key OOP concepts:

---

1. Classes and Objects

- **Class**: A blueprint or template for creating objects (instances). It defines a set of attributes and methods.
    - Example:

python

CopyEdit

class Dog:

```
    def __init__(self, name, age):

        self.name = name

        self.age = age


    def bark(self):

        return f"{self.name} barks!"
```

- **Object**: An instance of a class. Each object can hold different values for the attributes defined in the class.
    - Example:

python

CopyEdit

```
dog1 = Dog("Buddy", 3)

dog2 = Dog("Max", 2)
```

---

2. Encapsulation

Encapsulation is the concept of bundling data (attributes) and methods that operate on the data within one unit (a class). It also restricts direct access to some of an object's components, which can prevent unintended interference.

- **Private attributes**: Use an underscore (_) or double underscore (__) to mark attributes or methods as private.
    - Example:

python

CopyEdit

```
class Car:

    def __init__(self, brand, model):

        self._brand = brand  # protected

        self.__model = model  # private


    def get_model(self):

        return self.__model
```

---

3. Inheritance

Inheritance allows a new class (child class) to inherit attributes and methods from an existing class (parent class). It enables code reuse and the creation of hierarchical relationships.

- **Parent Class**: The class being inherited from.
- **Child Class**: The class that inherits from the parent class.
  - Example:

python

CopyEdit

```
class Animal:
    def sound(self):
        return "Animal sound"


class Dog(Animal):
    def sound(self):
        return "Bark"
```

---

4. Polymorphism

Polymorphism allows one method or operator to operate on different types of data. It enables a single function to be used for different types or classes.

- **Method Overriding**: A child class can provide a specific implementation of a method that is already defined in its parent class.
  - Example:

python

CopyEdit

```
class Animal:
    def sound(self):
        return "Animal sound"


class Dog(Animal):
    def sound(self):
        return "Bark"


dog = Dog()
```

```
print(dog.sound())  # Output: Bark
```

- **Method Overloading** (not directly supported in Python, but can be simulated using default arguments):

python

CopyEdit

```
class Calculator:
    def add(self, a, b=0):
        return a + b


calc = Calculator()
print(calc.add(5))     # Output: 5
print(calc.add(5, 3))  # Output: 8
```

---

5. Abstraction

Abstraction involves hiding the complex implementation details and showing only the essential features of an object. In Python, this is often achieved using abstract classes.

- **Abstract Class**: A class that cannot be instantiated directly, but can be subclassed. It may have abstract methods (methods that must be implemented by the child class).
    - Example:

python

CopyEdit

```
from abc import ABC, abstractmethod


class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass


class Dog(Animal):
    def sound(self):
        return "Bark"
```

---

6. Constructor and Destructor

- **Constructor (__init__)**: A special method that is called when an object is created. It initializes the object's attributes.
- **Destructor (__del__)**: A special method called when an object is destroyed (optional in Python).
    - Example:

python

CopyEdit

```
class Person:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} is created.")

    def __del__(self):
        print(f"{self.name} is destroyed.")

p = Person("John")
del p  # This will call the destructor
```

Summary of OOP Concepts:

1. **Classes and Objects**: Templates and instances of data and methods.
2. **Encapsulation**: Bundles data and restricts access to certain parts of an object.
3. **Inheritance**: Allows classes to inherit properties from other classes.
4. **Polymorphism**: Enables different methods or functions to work on different data types.
5. **Abstraction**: Hides implementation details and exposes only essential information.
6. **Constructor and Destructor**: Special methods for object creation and destruction.

Ruff

Ruff in Python – Meaning & Key Points

📌 **Meaning:**

Ruff is a fast, modern Python linter and formatter designed to enforce coding style and catch errors efficiently. It is an alternative to tools like **Flake8, Black, and isort**, but significantly faster as it is written in **Rust**.

🔷 **Key Points:**

1. **Fast & Lightweight** – Built with Rust, making it much faster than traditional Python linters.
2. **All-in-One Tool** – Combines linting, formatting, and import sorting in a single tool.
3. **Highly Configurable** – Supports custom rules, exclusions, and integrations.
4. **Flake8-Compatible** – Can replace multiple Flake8 plugins.
5. **Fixes Issues Automatically** – Can auto-correct common coding errors and style issues.
6. **Works with CI/CD Pipelines** – Ideal for automated code quality checks.

Logging

Logging in Python is the process of tracking events that happen during the execution of a program. The **logging** module provides a flexible way to record messages for debugging, monitoring, and auditing purposes.

🔷 **Key Points:**

1. **Built-in Module** – Python provides the logging module for easy logging.
2. **Different Log Levels** – Messages are categorized by severity:
    - DEBUG (Detailed info for debugging)
    - INFO (General information)
    - WARNING (Something unexpected but not critical)
    - ERROR (A problem that needs attention)
    - CRITICAL (Serious error, program might stop)
3. **Log Formatting** – Customize logs with timestamps, module names, etc.
4. **Multiple Handlers** – Logs can be written to files, consoles, or remote servers.
5. **Logging Instead of Print** – Recommended over print() for better control and debugging.
6. **Configuration** – Can be set up using logging.basicConfig() or a config file.

Git

Git is a distributed version control system (VCS) used for tracking changes in source code during software development. It allows multiple developers to work on a project simultaneously while keeping track of every modification.

1. **Version Control** – Tracks changes in code and allows reverting to previous versions.
2. **Distributed** – Each developer has their own local copy of the repo.
3. **Branching & Merging** – Supports creating branches and merging them back.
4. **Staging Area** – Changes are staged before committing.
5. **Commit** – Represents a snapshot of changes with a unique ID.
6. **Collaboration** – Enables working with others via pushing and pulling from a remote repo.
7. **Popular Platforms** – Works with GitHub, GitLab, Bitbucket, etc.